Enrolment No./Seat No_____

GUJARAT TECHNOLOGICAL UNIVERSITY

BE- SEMESTER–IV (NEW) EXAMINATION – WINTER 2024

Subject Code:3140705 Date:30-11-2024

Subject Name:Object Oriented Programming -I

Time:02:30 PM TO 05:00 PM Total Marks:70

Instructions:

1. **Attempt all questions.**
2. **Make suitable assumptions wherever necessary.**
3. **Figures to the right indicate full marks.**
4. **Simple and non-programmable scientific calculators are allowed.**

			Marks
Q.1	**(a)**	Why public, static, void and arguments in following statement required → public static void main(String args[])	**03**
	(b)	List out different types of loops available in java with code snippet.	**04**
	(c)	Describe Inheritance, Polymorphism and Data Encapsulation characteristics of OOP with example.	**07**
Q.2	**(a)**	State three use of final keyword.	**03**
	(b)	Describe java Access modifier.	**04**
	(c)	Define concept of method overloading and method overriding with examples.	**07**
		OR	
	(c)	What is Abstract class? Describe usage of Abstract class with one example.	**07**
Q.3	**(a)**	Describe use of Super keyword.	**03**
	(b)	Describe various methods and usage of Object class in java.	**04**
	(c)	Create a class called Employee that includes three pieces of information as instance variables—a first name (typeString), a last name (typeString) and a monthly salary (double). Your class should have a constructor that initializes the three instance variables. Provide a set and a get method for each instance variable. If the monthly salary is not positive, set it to 0.0. Write a test application named EmployeeTest that demonstrates class Employee's capabilities. Create two Employee objects and display each object's yearly salary. Then give each Employee a 10% raise and display each Employee's yearly salary again.	**07**
		OR	
Q.3	**(a)**	Describe use of static keyword.	**03**
	(b)	Define property of Dynamic method dispatch with example.	**04**
	(c)	Create a class named Account with instance variables Ac_No,Name and Balance and methods display (), setdata (),deposit(). In display () method, display the instance variables value Ac_No,Name and Balance). And in setdata () method set the instance variable values (Ac_No,Name and Balance) and in deposit() method the amount that user wants to deposit that will be deposited.	**07**
Q.4	**(a)**	Write a Java program to copy one array list into another.	**03**
	(b)	Read two numbers from command line argument and print sum of them.	**04**
	(c)	Describe steps to create a package with one example.	**07**

OR

Q.4	**(a)**	Describe following keywords: (i) throw (ii) throws	**03**
	(b)	Compare Abstract class and Interface	**04**
	(c)	Create a method named Withdraw () in the main function performing exception handling. Example: Balance = 1000; Withdraw = 12000 Here, Balance<Withdraw (throw exception)	**07**
Q.5	**(a)**	Describe following term : finally and finalize	**03**
	(b)	Write a java program to read employee details from emp.txt file and print on screen.	**04**
	(c)	Create customer class which extends thread class and contains two instance variables name, ProductName and static variable Product Quantity. For example if two customers are trying to buy the same product at once then follow synchronization of two customers extending thread class.	**07**

OR

Q.5	**(a)**	Describe any one wrapper class with its method usage.	**03**
	(b)	Write a java program to read students details from console and write that students details into emp.txt file.	**04**
	(c)	Write a program to create GUI for Student registration form using JavaFX.	**07**

****

Object Oriented Programming –I

WINTER 2024

Q.1 (a) Why public, static, void and arguments in following statement required
→ public static void main(String args[])

1. public

- The word **public** means the method is **accessible to everyone**.
- The JVM (Java Virtual Machine) must call the `main()` method from **outside** the class, so it must be **public**.
- If it is not public, JVM cannot access it and your program will not start.

---

2. static

- The `main()` method is declared **static** so that it can run **without creating an object** of the class.
- JVM needs to call `main()` directly when the program starts; it cannot create objects automatically.
- Being static ensures the method belongs to the class, not to any object.

---

3. void

- `void` specifies that the `main()` method **does not return any value**.
- Since the JVM only needs to execute your program, it does not expect any output from the `main()` method.
- Therefore, the return type is void.

---

4. main

- `main` is the **starting point** of every Java program.
- JVM searches for this method as the entry point to begin execution.

---

5. String args[]

- It accepts **command-line arguments**, which are values passed to the program during execution.
- These arguments are stored in an array of **String type**.
- Example:
  If you run → `java Test Hello`,
  then `args[0] = "Hello"`.

(b) List out different types of loops available in java with code snippet.

1. For Loop

Definition:
A `for` loop is used when the number of iterations is known in advance. It consists of three parts: initialization, condition, and increment/decrement. The loop continues executing the code block as long as the condition is true. It is commonly used for counting loops or iterating over arrays with an index.

Example:

```
for(int i = 1; i <= 5; i++) {
    System.out.println("Iteration number: " + i);
}
```

Output:

```
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
```

---

2. While Loop

Definition:
A `while` loop is used when the number of iterations is not known in advance. The loop checks the condition before each iteration. If the condition is true, the loop executes; if false, the loop stops. It is ideal for situations where the loop depends on a dynamic condition.

Example:

```
int i = 1;
while(i <= 5) {
    System.out.println("Iteration number: " + i);
    i++;
}
```

Output:

```
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
```

---

3. Do-While Loop

Definition:
A `do-while` loop is similar to a `while` loop, but the condition is checked after the loop executes. This ensures that the loop body is executed at least once, even if the condition is initially false. It is useful when the code inside the loop must run before the condition is tested.

Example:

```
int i = 1;
do {
    System.out.println("Iteration number: " + i);
    i++;
} while(i <= 5);
```

Output:

```
Iteration number: 1
Iteration number: 2
Iteration number: 3
Iteration number: 4
Iteration number: 5
```

---

4. Enhanced For Loop (For-Each Loop)

Definition:
The enhanced `for` loop, also called a for-each loop, is used to iterate over elements of arrays or collections without using an index. It is simpler and cleaner for iterating over every element in a data structure.

Example:

```
int[] numbers = {10, 20, 30, 40, 50};
for(int num : numbers) {
    System.out.println("Number: " + num);
}
```

Output:

```
Number: 10
Number: 20
Number: 30
Number: 40
Number: 50
```

(c) Describe Inheritance, Polymorphism and Data Encapsulation characteristics of OOP with example.

---

1. Inheritance

Definition:
Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP). It allows a class (known as the **child class** or **subclass**) to acquire the properties and methods of another class (called the **parent class** or **superclass**). This helps in **code reusability**, reduces redundancy, and enables hierarchical classification of classes. Using inheritance, a child class can add its own features or modify the existing features of the parent class.

Key Points:

- Promotes **code reuse**.
- Establishes **IS-A relationship** between parent and child classes.
- Supports **method overriding** in combination with polymorphism.

Example:

```
// Parent class
class Vehicle {
    String brand = "Ford";
    void honk() {
        System.out.println("Beep Beep!");
    }
}

// Child class inherits Vehicle
class Car extends Vehicle {
    String model = "Mustang";
    void display() {
        System.out.println(brand + " " + model);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.honk();         // inherited method
        myCar.display();      // child class method
    }
}
```

Output:

```
Beep Beep!
Ford Mustang
```

---

2. Polymorphism

Definition:
Polymorphism means "**many forms**." It allows a single entity, like a method or an object, to behave differently in different situations. Polymorphism in Java is mainly of two types:

1. **Compile-time Polymorphism (Method Overloading):** Same method name with different parameters (number, type, or order). The compiler decides which method to call.
2. **Run-time Polymorphism (Method Overriding):** A child class provides its own implementation of a method that exists in the parent class. The method to call is decided at runtime.

Advantages:

- Improves **flexibility** and **maintainability**.
- Supports **dynamic method invocation**.
- Helps in **implementing real-world scenarios** where a single action can behave differently.

Example (Method Overriding – Run-time Polymorphism):

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal1 = new Dog(); // runtime polymorphism
        Animal myAnimal2 = new Cat();
        myAnimal1.sound();
        myAnimal2.sound();
    }
}
```

Output:

```
Dog barks
Cat meows
```

---

3. Data Encapsulation

Definition:
Data encapsulation is the process of **wrapping data (variables) and methods** into a single unit called a class. The **data is hidden** from outside access using access modifiers (usually `private`) and can only be accessed or modified using **public getter and setter methods**. This provides **data security**, prevents unauthorized modification, and allows controlled access to class members.

Key Points:

- Protects **data integrity**.
- Hides internal implementation (**data hiding**).
- Allows changes to be made in class implementation without affecting outside code.

Example:

```
class Student {
    private String name; // private variable

    // Setter method
    public void setName(String n) {
        name = n;
    }

    // Getter method
    public String getName() {
        return name;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.setName("Komal");   // controlled access
        System.out.println(s.getName());
    }
}
```

Output:

```
Komal
```

Q.2 (a) State three use of final keyword.

`final` keyword in Java is used to **declare constants, prevent inheritance, and prevent method overriding**. Once a `final` entity is assigned or defined, it **cannot be changed or modified**.

Three Uses of `final` Keyword:

1. **Final Variable (Constant):**
   - When a variable is declared as `final`, its value **cannot be changed** once it is initialized.
   - It is used to define **constants** in a program.

Example:

```
final int MAX_VALUE = 100;
// MAX_VALUE = 200; // Error: cannot change final variable
System.out.println("Max Value: " + MAX_VALUE);
```

2. **Final Method:**
   - When a method is declared as `final`, it **cannot be overridden** by subclasses.
   - This ensures that the implementation of the method remains **unchanged** in child classes.

Example:

```
class Parent {
    final void showMessage() {
        System.out.println("This is a final method.");
    }
}

class Child extends Parent {
    // void showMessage() {} // Error: cannot override final method
}
```

3. **Final Class:**
   - When a class is declared as `final`, it **cannot be inherited**.
   - This is used to **prevent extension** of the class for security or design reasons.

Example:

```
final class Vehicle {
    void display() {
        System.out.println("This is a final class.");
    }
}

// class Car extends Vehicle {} // Error: cannot inherit from final
class
```

(b) Describe java Access modifier.

Access modifiers in Java are **keywords** used to set the **visibility or accessibility of classes, methods, and variables**. They control **which parts of a program can access certain members** of a class, helping in **data hiding and encapsulation**.

Java has **four types of access modifiers**:

---

1. Private (`private`)

- Members declared `private` are accessible **only within the same class**.
- It provides the **highest level of data protection**.

Example:

```
class Student {
    private String name = "Komal";

    private void display() {
        System.out.println("Name: " + name);
    }

    public void show() {
        display(); // accessible within class
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.show(); // OK
        // s.display(); // Error: private method not accessible outside
class
    }
}
```

---

2. Default (Package-Private) [No Modifier]

- If no modifier is specified, it is **default**.
- Members are accessible **only within the same package**.
- It is **more restrictive than protected or public**, but less than private.

Example:

```
class Student {
    String name = "Komal"; // default variable

    void display() {
        System.out.println("Name: " + name);
    }
}
```

- Accessible by other classes in the same package but **not from other packages**.

---

3. Protected (`protected`)

- Members declared `protected` are accessible **within the same package** and by **subclasses (even in different packages)**.

- Useful in inheritance for sharing properties safely.

Example:

```
class Vehicle {
    protected String type = "Car";
}

class Car extends Vehicle {
    void showType() {
        System.out.println("Vehicle type: " + type); // accessible in
subclass
    }
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.showType();
    }
}
```

---

4. Public (`public`)

- Members declared `public` are accessible **from anywhere** in the program.
- Offers **no restriction**, so use carefully to maintain encapsulation.

Example:

```
class Student {
    public String name = "Komal";

    public void display() {
        System.out.println("Name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        s.display(); // accessible anywhere
    }
}
```

(c) Define concept of method overloading and method overriding with examples.

1. Method Overloading (Compile-Time Polymorphism)

Definition:
Method overloading occurs when **two or more methods in the same class have the same name but different parameters** (different type, number, or order of parameters). It is also

called **compile-time polymorphism** because the compiler decides which method to call based on the arguments.

Key Points:

- Same method name, **different parameter list**.
- Return type **can be same or different**.
- Occurs **within the same class**.
- Used to increase **readability and reusability** of code.

Example:

```
class Calculator {
    // Method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two doubles
    double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(10, 20));        // Calls first method
        System.out.println(calc.add(10, 20, 30));    // Calls second method
        System.out.println(calc.add(5.5, 4.5));      // Calls third method
    }
}
```

Output:

```
30
60
10.0
```

---

2. Method Overriding (Run-Time Polymorphism)

Definition:
Method overriding occurs when a **subclass provides its own implementation of a method that is already defined in its superclass**. It is also called **run-time polymorphism** because the method to call is determined at **runtime** based on the object type.

Key Points:

- Same method name, **same parameter list**.

- Return type must be **same or covariant**.
- Access level in subclass **cannot be more restrictive** than in superclass.
- Used to **modify or extend the behavior** of inherited methods.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a1 = new Animal();
        Animal a2 = new Dog(); // runtime polymorphism
        a1.sound(); // calls superclass method
        a2.sound(); // calls subclass method
    }
}
```

Output:

```
Animal makes a sound
Dog barks
```

OR

(c) What is Abstract class? Describe usage of Abstract class with one example.

An **abstract class** in Java is a class that **cannot be instantiated directly** and may contain **abstract methods** (methods without a body) as well as **concrete methods** (methods with a body). Abstract classes are used to provide a **base structure for subclasses** and enforce a common behavior while allowing the subclasses to implement specific details.

Key Points:

- Declared using the `abstract` keyword.
- Can contain **abstract methods** and **regular methods**.
- Cannot create objects of an abstract class directly.
- A subclass **must implement all abstract methods** of the abstract class or itself become abstract.
- Supports **partial abstraction** (unlike interfaces which are full abstraction in older Java versions).

Usage of Abstract Class:

1. **Provide a common template** for related classes.
2. **Define some methods with common implementation** and leave others for subclasses to implement.
3. Helps in **achieving abstraction** and **polymorphism** in OOP.
4. Encourages **code reusability** by allowing common methods to be defined in the abstract class.

Example:

```
// Abstract class
abstract class Shape {
    String color;

    // Abstract method (no body)
    abstract void area();

    // Concrete method
    void setColor(String c) {
        color = c;
    }
}

// Subclass
class Circle extends Shape {
    double radius;

    Circle(double r, String c) {
        radius = r;
        setColor(c);
    }

    // Implementing abstract method
    @Override
    void area() {
        double area = 3.14 * radius * radius;
        System.out.println("Circle area: " + area + ", Color: " + color);
    }
}

public class Main {
    public static void main(String[] args) {
        // Shape s = new Shape(); // Error: cannot instantiate abstract
class
        Circle c1 = new Circle(5, "Red");
        c1.area();
    }
}
```

Output:

```
Circle area: 78.5, Color: Red
```

Q.3 (a) Describe use of Super keyword.

The `super` keyword in Java is a **reference variable** used in a subclass to refer to its **immediate parent class**. It is commonly used to access **parent class members (variables, methods, and constructors)** from a subclass. The `super` keyword helps in **avoiding name conflicts** and supports **inheritance**.

Uses of `super` Keyword:

1. **Access Parent Class Variables:**
   - When a subclass has a variable with the same name as the parent class, `super` is used to refer to the **parent class variable**.
   - Helps in **resolving name conflicts** between parent and child class members.
2. **Call Parent Class Methods:**
   - If a subclass **overrides a method** of the parent class, `super` can be used to **invoke the parent class version** of that method.
   - Ensures that the original behavior of the parent method can still be accessed.
3. **Invoke Parent Class Constructor:**
   - `super()` can be used in a subclass constructor to **call a constructor of the parent class**.
   - This is useful to **initialize parent class members** before initializing the child class.
4. **Supports Inheritance and Code Reusability:**
   - By using `super`, subclasses can **reuse the properties and behaviors** of parent classes without rewriting code.
   - Maintains a **clear hierarchical relationship** in object-oriented programs.

Example:

```
class Parent {
   int num = 100;
}

class Child extends Parent {
   int num = 200;

   void showNumbers() {
      System.out.println("Child num: " + num);      // Child variable
      System.out.println("Parent num: " + super.num); // Parent variable
   }
}
```

(b) Describe various methods and usage of Object class in java.

In Java, **`Object` class** is the **root class of all classes**. Every class in Java **directly or indirectly inherits** from `Object`. This means all objects in Java are subclasses of `Object`. The `Object` class provides **common methods** that can be used by all objects, which ensures basic functionalities like comparison, cloning, string representation, etc.

Key Methods of Object Class and Their Usage:

1. **`toString()`**
   - Returns a **string representation of the object**.
   - By default, it returns the class name followed by the object's hashcode.
   - Can be **overridden** to provide meaningful string output.
2. **`equals(Object obj)`**
   - Compares **two objects for equality**.
   - By default, it checks **reference equality** (whether both references point to the same object).
   - Can be **overridden** to compare objects based on their **content or state**.
3. **`hashCode()`**
   - Returns an **integer hash code value** for the object.
   - Used in **hash-based collections** like `HashMap`, `HashSet`.
   - When `equals()` is overridden, `hashCode()` should also be overridden.
4. **`getClass()`**
   - Returns the **runtime class** of the object.
   - Useful for **reflection**, type checking, and debugging.
5. **`clone()`**
   - Creates and returns a **copy of the object**.
   - The class must implement the `Cloneable` interface to use this method.
6. **`finalize()`**
   - Called by the **garbage collector** before an object is destroyed.
   - Can be used to release resources, but its use is discouraged in modern Java.
7. **`notify()`, `notifyAll()`, and `wait()`**
   - Methods used in **multithreading** for inter-thread communication.
   - `wait()` makes a thread wait, `notify()` wakes up one waiting thread, and `notifyAll()` wakes up all waiting threads.

(c) Create a class called Employee that includes three pieces of information as instance variables—a first name (typeString), a last name (typeString) and a monthly salary (double). Your class should have a constructor that initializes the three instance variables. Provide a set and a get method for each instance variable. If the monthly salary is not positive, set it to 0.0. Write a test application named EmployeeTest that demonstrates class Employee's capabilities. Create two Employee objects and display each object's yearly salary. Then give each Employee a 10% raise and display each Employee's yearly salary again.

Employee.java

```
public class Employee {

   // Instance variables

   private String firstName;
```

```
   private String lastName;
   private double monthlySalary;
// Constructor
   public Employee(String firstName, String lastName, double monthlySalary) {
      this.firstName = firstName;
      this.lastName = lastName;
      if(monthlySalary > 0.0) {
         this.monthlySalary = monthlySalary;
      } else {
         this.monthlySalary = 0.0;
      }
   }
// Getter and Setter for firstName
   public String getFirstName() {
      return firstName;
   }
public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
// Getter and Setter for lastName
   public String getLastName() {
      return lastName;
   }
public void setLastName(String lastName) {
      this.lastName = lastName;
   }
// Getter and Setter for monthlySalary
   public double getMonthlySalary() {
      return monthlySalary;
```

```
   }
   public void setMonthlySalary(double monthlySalary) {
      if(monthlySalary > 0.0) {
         this.monthlySalary = monthlySalary;
      } else {
         this.monthlySalary = 0.0;
      }
   }// Method to calculate yearly salary
   public double getYearlySalary() {
      return monthlySalary * 12;
   }
// Method to give a raise in percentage
   public void giveRaise(double percent) {
      if(percent > 0) {
         monthlySalary += monthlySalary * percent / 100;
      }
   }
}
```

EmployeeTest.java

```
public class EmployeeTest {
   public static void main(String[] args) {
      // Create two Employee objects
      Employee emp1 = new Employee("Anit", "Javiya", 5000);
      Employee emp2 = new Employee("Nisha", "Kapoor", 6000);
      // Display yearly salary before raise
      System.out.println(emp1.getFirstName() + " " + emp1.getLastName() +
                         "'s yearly salary: " + emp1.getYearlySalary());
      System.out.println(emp2.getFirstName() + " " + emp2.getLastName() +
```

```
                "'s yearly salary: " + emp2.getYearlySalary());

        // Give 10% raise to both employees
        emp1.giveRaise(10);
        emp2.giveRaise(10);

        // Display yearly salary after raise
        System.out.println("\nAfter 10% raise:");
        System.out.println(emp1.getFirstName() + " " + emp1.getLastName() +
                "'s yearly salary: " + emp1.getYearlySalary());
        System.out.println(emp2.getFirstName() + " " + emp2.getLastName() +
                "'s yearly salary: " + emp2.getYearlySalary());
    }
}
```

OUTPUT:'

```
Anit Javiya's yearly salary: 60000.0
Nisha Kapoor's yearly salary: 72000.0

After 10% raise:
Anit Javiya's yearly salary: 66000.0
Nisha Kapoor's yearly salary: 79200.0
```

OR

Q.3 (a) Describe use of static keyword.

The `static` keyword in Java is used to **declare class-level members**, which means the variable, method, or block belongs to the **class itself** rather than to any particular instance (object) of the class. Static members are **shared across all objects** of the class.

Uses of `static` Keyword:

1. **Static Variables (Class Variables):**
   - A `static` variable is **shared by all instances** of a class.

    - It retains the same value for all objects and is initialized **once at class loading time**.
    - Useful for **constants or counters** that should be common across all objects.
2. **Static Methods (Class Methods):**
    - A `static` method **belongs to the class** and can be called **without creating an object**.
    - It **cannot access instance variables or methods directly**; it can only access static members.
    - Commonly used for **utility or helper methods**, e.g., `Math.sqrt()` or `Integer.parseInt()`.
3. **Static Blocks (Static Initialization Block):**
    - Used to **initialize static variables** at the time of class loading.
    - Executes **only once**, before any object is created or any static method is called.
4. **Static Nested Classes:**
    - A **nested class** can be declared static.
    - A static nested class **does not require an instance** of the outer class to be created.

Advantages of Static Keyword:

- **Memory efficiency:** Only one copy of static members is stored in memory, shared by all objects.
- **Convenience:** Static methods can be called without creating objects.
- **Consistency:** Static variables maintain a common value across all objects.

(b) Define property of Dynamic method dispatch with example.

Dynamic Method Dispatch is a mechanism in Java by which a **call to an overridden method is resolved at runtime** rather than compile time. It allows a **superclass reference variable** to refer to a **subclass object**, and the **actual method that gets executed depends on the object type**, not the reference type.

It is a key feature of **runtime polymorphism** in Java.

Properties of Dynamic Method Dispatch (DMD):

1. **Superclass Reference Can Point to Subclass Object:**
    - In DMD, a reference variable of a **superclass** can refer to an object of any of its **subclasses**.
    - This allows **flexibility** in handling objects of different types using a common interface (the superclass reference).
2. **Method Overriding is Essential:**
    - Only **overridden methods** participate in dynamic method dispatch.
    - If a method is **not overridden** in the subclass, the **superclass version** of the method is called.
3. **Decision Made at Runtime:**

- The **actual method that gets executed** is determined **during program execution** (runtime) based on the **type of object**, not the reference type.
- This is why it is also called **runtime polymorphism**.

4. **Supports Polymorphism:**
   - DMD is a key feature of **polymorphism** in Java.
   - It allows **writing generic code** that can work with different object types while invoking the correct overridden methods automatically.
5. **Cannot Use with Static, Private, or Final Methods:**
   - Static methods are **class-level**, so they are resolved at **compile-time**, not runtime.
   - Private and final methods **cannot be overridden**, so DMD does not apply to them.
6. **Enables Loose Coupling:**
   - Code that uses superclass references to interact with subclass objects is **less dependent on specific subclass implementations**.
   - This makes the code **more flexible and maintainable**.
7. **Helps in Dynamic Behavior:**
   - The behavior of an object can **change dynamically** at runtime based on the actual type of the object assigned to the superclass reference.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a;       // superclass reference

        a = new Dog();  // reference refers to Dog object
        a.sound();      // calls Dog's sound() method

        a = new Cat();  // reference refers to Cat object
        a.sound();      // calls Cat's sound() method
    }
}
```

Output:

```
Dog barks
Cat meows
```

(c) Create a class named Account with instance variables Ac_No,Name and Balance and methods display (), setdata (),deposit(). In display () method, display the instance variables value Ac_No,Name and Balance). And in setdata () method set the instance variable values (Ac_No,Name and Balance) and in deposit() method the amount that user wants to deposit that will be deposited.

Account.java:

```
import java.util.Scanner;

public class Account {

   // Instance variables

   private int Ac_No;

   private String Name;

   private double Balance;


   // Method to set account data

   public void setData(int acNo, String name, double balance) {

      this.Ac_No = acNo;

      this.Name = name;

      if(balance > 0) {

         this.Balance = balance;

      } else {

         this.Balance = 0.0;

      }

   }


   // Method to deposit amount

   public void deposit(double amount) {

      if(amount > 0) {
```

```
            Balance += amount;
            System.out.println("Amount " + amount + " deposited successfully.");
        } else {
            System.out.println("Invalid amount. Deposit failed.");
        }
    }

    // Method to display account details
    public void display() {
        System.out.println("Account Number: " + Ac_No);
        System.out.println("Account Holder Name: " + Name);
        System.out.println("Account Balance: " + Balance);
    }
}
```

AccountTest.java:

```
import java.util.Scanner;

public class AccountTest {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Create an Account object
        Account acc = new Account();

        // Set account data
        System.out.print("Enter Account Number: ");
        int acNo = sc.nextInt();
        sc.nextLine(); // consume newline
        System.out.print("Enter Account Holder Name: ");
```

```
        String name = sc.nextLine();
        System.out.print("Enter Initial Balance: ");
        double balance = sc.nextDouble();

        acc.setData(acNo, name, balance);

        // Display account details
        System.out.println("\nAccount Details:");
        acc.display();

        // Deposit amount
        System.out.print("\nEnter amount to deposit: ");
        double amount = sc.nextDouble();
        acc.deposit(amount);

        // Display updated account details
        System.out.println("\nUpdated Account Details:");
        acc.display();

        sc.close();
    }
}
```

Output:

```
Enter Account Number: 101
Enter Account Holder Name: Komal
Enter Initial Balance: 5000

Account Details:
```

```
Account Number: 101
Account Holder Name: Komal
Account Balance: 5000.0

Enter amount to deposit: 1500
Amount 1500.0 deposited successfully.

Updated Account Details:
Account Number: 101
Account Holder Name: Komal
Account Balance: 6500.0
```

Q.4 (a) Write a Java program to copy one array list into another.

```
import java.util.ArrayList;

public class CopyArrayList {
    public static void main(String[] args) {
        // Original ArrayList
        ArrayList<String> list1 = new ArrayList<>();
        list1.add("Apple");
        list1.add("Banana");
        list1.add("Orange");

        // Copying list1 into list2
        ArrayList<String> list2 = new ArrayList<>(list1);

        // Display both lists
```

```
            System.out.println("Original List: " + list1);
            System.out.println("Copied List: " + list2);
        }
    }
```

OUTPUT:

```
Original List: [Apple, Banana, Orange]
Copied List: [Apple, Banana, Orange]
```

(b) Read two numbers from command line argument and print sum of them.

```
public class SumFromCommandLine {
    public static void main(String[] args) {
        if(args.length < 2) {
            System.out.println("Please provide two numbers as command line arguments.");
            return;
        }
  // Read numbers from command line arguments
        int num1 = Integer.parseInt(args[0]);
        int num2 = Integer.parseInt(args[1]);

        // Calculate sum
        int sum = num1 + num2;

        // Display result
        System.out.println("Sum of " + num1 + " and " + num2 + " is: " + sum);
    }
}
```

How to Run:

Compile the program:

```
javac SumFromCommandLine.java
```

Run with two numbers:

```
java SumFromCommandLine 10 20
```

Output:

```
Sum of 10 and 20 is: 30
```

(c) Describe steps to create a package with one example.

A **package** in Java is a **namespace that organizes classes and interfaces**. Packages help in:

- Avoiding **name conflicts**.
- **Organizing classes** logically.
- **Reusing code** across projects.

Steps to Create and Use a Package:

Step 1: Create a Package

1. Choose a name for your package (e.g., `mypackage`).
2. At the top of your Java file, use the **`package` keyword** followed by the package name.

```
// File: MyClass.java
package mypackage;

public class MyClass {
    public void display() {
        System.out.println("Hello from MyClass in mypackage!");
    }
}
```

3. Save the file in a **folder with the same name as the package** (`mypackage`).

Step 2: Compile the Package

- Open the terminal or command prompt.
- Navigate to the **parent folder** of `mypackage`.
- Compile the Java file:

```
javac mypackage/MyClass.java
```

- This creates a **`MyClass.class`** file inside the `mypackage` folder.

Step 3: Use the Package in Another Program

1. Create another Java file (e.g., `TestPackage.java`) **outside the package folder**.
2. Import the package using the `import` keyword.

```
// File: TestPackage.java
import mypackage.MyClass;

public class TestPackage {
    public static void main(String[] args) {
        MyClass obj = new MyClass(); // Create object of MyClass
        obj.display();               // Call method
    }
}
```

Step 4: Compile and Run the Program

1. Compile the program:

```
javac TestPackage.java
```

2. Run the program:

```
java TestPackage
```

Output:

```
Hello from MyClass in mypackage!
```

Q.4 (a) Describe following keywords: (i) throw (ii) throws .

(i) `throw` Keyword

Definition:

- The `throw` keyword in Java is used to **explicitly throw an exception** from a method or a block of code.
- It is used when a **specific condition arises** in the program and you want to **signal an exceptional situation**.

Key Points:

- Only **one exception object** can be thrown at a time.
- Syntax:

```
        throw new ExceptionType("Error Message");
```

- The exception must be **handled** using `try-catch` or declared in the method using `throws`.

Example:

```
int age = 15;
if(age < 18) {
    throw new ArithmeticException("Age must be 18 or above.");
}
```

Usage:

- To **raise exceptions intentionally** when a program encounters an invalid condition.

(ii) `throws` Keyword

Definition:

- The `throws` keyword in Java is used in a **method declaration** to **declare the exceptions that a method can throw**.
- It **notifies the caller** of the method that it must handle or propagate the exception.

Key Points:

- Can declare **multiple exceptions** separated by commas.
- Syntax:

```
void methodName() throws ExceptionType1, ExceptionType2
```

- It **does not throw an exception** itself; it just **declares it**.

Example:

```
void divide(int a, int b) throws ArithmeticException {
    int result = a / b; // may throw ArithmeticException
    System.out.println("Result: " + result);
}
```

Usage:

- To **propagate checked or unchecked exceptions** to the calling method.
- Helps in **exception handling delegation**.

(b) Compare Abstract class and Interface

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract**methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

(c) Create a method named Withdraw () in the main function performing exception handling. Example: Balance = 1000; Withdraw = 12000 Here, Balance

```
import java.util.Scanner;
```

```
// Custom exception class
class InsufficientBalanceException extends Exception {
    public InsufficientBalanceException(String message) {
        super(message);
    }
}

public class WithdrawExample {
    public static void main(String[] args) {
        double balance = 1000;  // Initial balance
        Scanner sc = new Scanner(System.in);

        System.out.println("Current Balance: " + balance);
        System.out.print("Enter amount to withdraw: ");
        double withdrawAmount = sc.nextDouble();

        try {
            balance = withdraw(balance, withdrawAmount); // Call withdraw
method
            System.out.println("Withdrawal successful. Remaining Balance: " +
balance);
        } catch (InsufficientBalanceException e) {
            System.out.println("Withdrawal failed: " + e.getMessage());
        }

        sc.close();
```

```
    }


    // Method to perform withdrawal
    public static double withdraw(double balance, double amount) throws
InsufficientBalanceException {
        if (amount > balance) {
            // Throw custom exception if amount exceeds balance
            throw new InsufficientBalanceException("Insufficient balance. Cannot
withdraw " + amount);
        }
        return balance - amount; // Deduct amount if sufficient balance
    }
}
```

Sample Output 1 (Withdrawal within balance):

```
Current Balance: 1000.0
Enter amount to withdraw: 500
Withdrawal successful. Remaining Balance: 500.0
```

Sample Output 2 (Withdrawal exceeds balance):

```
Current Balance: 1000.0
Enter amount to withdraw: 12000
Withdrawal failed: Insufficient balance. Cannot withdraw 12000.0
```

Q.5 (a) Describe following term : finally and finalize

`finally` Block

Definition:

- The `finally` block in Java is used in **exception handling** to **execute a block of code regardless of whether an exception occurs or not**.
- It is **optional** but commonly used to **release resources** like files, database connections, or network sockets.

Key Points:

- It always **executes after the try-catch blocks**.

- Even if there is a `return` statement in `try` or `catch`, the `finally` block **still executes**.
- Typically used for **cleanup operations**.

`finalize()` Method

Definition:

- `finalize()` is a **method of the `Object` class** in Java.
- It is called by the **garbage collector** before an object is **destroyed** to perform cleanup operations.

Key Points:

- It allows an object to **release resources** before memory is reclaimed.
- You can **override `finalize()`** in your class to perform custom cleanup.
- **Not guaranteed** to run immediately when an object becomes unreachable; it runs **before garbage collection**, which is **nondeterministic**.
- Its usage is **discouraged in modern Java**; better to use `try-with-resources` or explicit resource management.

(b) Write a java program to read employee details from emp.txt file and print on screen.

emp.txt

101,Komal,50000

102,Anita,60000

103,Ankit,45000

Format: EmployeeID,Name,Salary

ReadEmployee.java

```
import java.io.*;

public class ReadEmployee {
    public static void main(String[] args) {
        String fileName = "emp.txt"; // File name
        String line;

        try {
            // Create FileReader and BufferedReader
            FileReader fr = new FileReader(fileName);
            BufferedReader br = new BufferedReader(fr);

            System.out.println("Employee Details:");
            System.out.println("-----------------");

            // Read each line from the file
            while ((line = br.readLine()) != null) {
```

```
                String[] parts = line.split(","); // Split line by comma
                int id = Integer.parseInt(parts[0]);
                String name = parts[1];
                double salary = Double.parseDouble(parts[2]);

                // Print employee details
                System.out.println("Employee ID: " + id);
                System.out.println("Name: " + name);
                System.out.println("Salary: " + salary);
                System.out.println("-----------------");
            }

            br.close(); // Close the BufferedReader
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + fileName);
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
}
```

Sample Output

```
Employee Details:
-----------------
Employee ID: 101
Name: Komal
Salary: 50000.0
-----------------
Employee ID: 102
Name: Anita
Salary: 60000.0
-----------------
Employee ID: 103
Name: Ankit
Salary: 45000.0
-----------------
```

(c) Create customer class which extends thread class and contains two instance variables name, ProductName and static variable Product Quantity. For example if two customers are trying to buy the same product at once then follow synchronization of two customers extending thread class.

Customer.java

```
class Customer extends Thread {
    private String name;
    private String productName;
    private static int productQuantity = 5; // Shared product quantity

    // Constructor
    public Customer(String name, String productName) {
        this.name = name;
        this.productName = productName;
    }

    // Synchronized method to buy product
    public static synchronized void buyProduct(String customerName) {
        if (productQuantity > 0) {
            System.out.println(customerName + " bought 1 product.
Remaining: " + (productQuantity - 1));
```

```
            productQuantity--; // Reduce product quantity
        } else {
            System.out.println(customerName + " cannot buy. Product out of
stock!");
        }
    }

    // Run method
    @Override
    public void run() {
        buyProduct(name); // Call synchronized method
    }
}
```

CustomerTest.java

```
public class CustomerTest {
    public static void main(String[] args) {
        // Create two customer threads trying to buy the same product
        Customer c1 = new Customer("Arya", "Laptop");
        Customer c2 = new Customer("Grishma", "Laptop");
        Customer c3 = new Customer("Ankit", "Laptop");

        // Start threads
        c1.start();
        c2.start();
        c3.start();
    }
}
```

Sample Output (order may vary due to thread scheduling)

```
Arya bought 1 product. Remaining: 4
Grishma bought 1 product. Remaining: 3
Ankit bought 1 product. Remaining: 2
```

If more customers try to buy than the available quantity:

```
Arya bought 1 product. Remaining: 0
Grishma cannot buy. Product out of stock!
Ankit cannot buy. Product out of stock!
```

OR

Q.5 (a) Describe any one wrapper class with its method usage.

Wrapper Class: `Integer`

Definition:

- The `Integer` class is a **wrapper class** in Java that wraps the **primitive `int` type** into an **object**.
- Wrapper classes allow **primitive types** to be used as **objects**, which is required in **collections**, **generics**, and **object-oriented features**.

Key Methods of `Integer` Class and Usage:

1. **`parseInt(String s)`**
   - Converts a string to an **int**.
   - Example:

```
int num = Integer.parseInt("123");  // num = 123
```

2. **`valueOf(String s)`**

- Converts a string to an **Integer object**.
- Example:

```
Integer obj = Integer.valueOf("456"); // obj = 456
```

3. **`intValue()`**

- Converts an **Integer object** to primitive **int**.
- Example:

```
Integer obj = 100;
int n = obj.intValue(); // n = 100
```

4. **`toString()`**

- Converts an **Integer object** or primitive int to **String**.
- Example:

```
int n = 50;
String str = Integer.toString(n); // str = "50"
```

5. **`compareTo(Integer anotherInteger)`**

- Compares two Integer objects and returns:
  - `0` if equal
  - Positive if first > second
  - Negative if first < second
- Example:

```
Integer a = 10, b = 20;
int result = a.compareTo(b); // result = -1
```

6. **`MAX_VALUE` and `MIN_VALUE`**

- Constants representing **maximum** and **minimum** values of int type.
- Example:

```
System.out.println("Max int: " + Integer.MAX_VALUE);
System.out.println("Min int: " + Integer.MIN_VALUE);
```

(b) Write a java program to read students details from console and write that students details into emp.txt file.

```
import java.io.*;
import java.util.Scanner;


public class StudentWrite {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String fileName = "emp.txt"; // File to write data


        try {
            FileWriter fw = new FileWriter(fileName, true); //
true for append mode
            BufferedWriter bw = new BufferedWriter(fw);
            PrintWriter pw = new PrintWriter(bw);


            System.out.print("Enter number of students: ");
            int n = sc.nextInt();
            sc.nextLine(); // consume newline


            for (int i = 0; i < n; i++) {
                System.out.println("Enter details of student "
+ (i + 1) + ":");


                System.out.print("ID: ");
                int id = sc.nextInt();
```

```
                sc.nextLine(); // consume newline

                System.out.print("Name: ");
                String name = sc.nextLine();

                System.out.print("Marks: ");
                double marks = sc.nextDouble();
                sc.nextLine(); // consume newline

                // Write student details to file in CSV format
                pw.println(id + "," + name + "," + marks);
            }

            pw.close();
            System.out.println("Student details written to " +
fileName + " successfully.");
        } catch (IOException e) {
            System.out.println("Error writing to file: " +
e.getMessage());
        }

        sc.close();
    }
}
```

Sample Input

```
Enter number of students: 2
Enter details of student 1:
ID: 101
Name: Alia
Marks: 85
Enter details of student 2:
ID: 102
Name: Anika
Marks: 90
```

Contents of emp.txt after execution

```
101, Alia, 85.0
102, Anika ,90.0
```

(c) Write a program to create GUI for Student registration form using JavaFX.

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class StudentRegistrationForm extends Application {

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Student Registration Form");

        // Create a GridPane layout
        GridPane grid = new GridPane();
        grid.setPadding(new Insets(20));
        grid.setHgap(10);
        grid.setVgap(10);

        // Labels and TextFields
        Label lblName = new Label("Name:");
        TextField txtName = new TextField();
        Label lblID = new Label("Student ID:");
        TextField txtID = new TextField();
        Label lblEmail = new Label("Email:");
        TextField txtEmail = new TextField();
        Label lblCourse = new Label("Course:");
        TextField txtCourse = new TextField();
        Label lblAge = new Label("Age:");
        TextField txtAge = new TextField();

        // Button
        Button btnSubmit = new Button("Submit");
        Label lblMessage = new Label();

        // Add components to the grid
        grid.add(lblName, 0, 0);
        grid.add(txtName, 1, 0);
        grid.add(lblID, 0, 1);
        grid.add(txtID, 1, 1);
        grid.add(lblEmail, 0, 2);
        grid.add(txtEmail, 1, 2);
        grid.add(lblCourse, 0, 3);
        grid.add(txtCourse, 1, 3);
```

```
        grid.add(lblAge, 0, 4);
        grid.add(txtAge, 1, 4);
        grid.add(btnSubmit, 1, 5);
        grid.add(lblMessage, 1, 6);

        // Button action
        btnSubmit.setOnAction(e -> {
            String name = txtName.getText();
            String id = txtID.getText();
            String email = txtEmail.getText();
            String course = txtCourse.getText();
            String age = txtAge.getText();

            if (name.isEmpty() || id.isEmpty() ||
email.isEmpty() || course.isEmpty() || age.isEmpty()) {
                lblMessage.setText("Please fill all fields!");
            } else {
                lblMessage.setText("Registration
Successful!\n" +
                        "Name: " + name + "\n" +
                        "ID: " + id + "\n" +
                        "Email: " + email + "\n" +
                        "Course: " + course + "\n" +
                        "Age: " + age);
            }
        });

        // Create scene and show stage
        Scene scene = new Scene(grid, 400, 350);
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```